Announcement

• If you are enrolled to the class, but have not received the email from Piazza, please send me an email

• Note: Monday 9/5 is a holiday – no office hour
• Jung will hold an additional office hour on 9/6, Tuesday, 1 pm, North N303B
• Guest lecture by Jung on 9/7 (Wed) on Map Reduce and Spark
  – Will be useful to start HW2
  – Work on HW1 and HW2 at your own pace, but note the deadlines!
Today’s topic

• SQL in a nutshell
• Reading material
  – [RG] Chapters 3 and 5
  – Additional reading for practice
    [GUW] Chapter 6
• Try SQL from today’s lecture on a toy dataset
  or DBLP dataset on PostGres!

Acknowledgement:
The following slides have been created adapting the
instructor material of the [RG] book provided by the authors
Dr. Ramakrishnan and Dr. Gehrke.
Relational Query Languages

• A major strength of the relational model: supports simple, powerful *querying* of data.

• Queries can be written intuitively, and the DBMS is responsible for an efficient evaluation
  – The key: precise semantics for relational queries.
  – Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.
The SQL Query Language

• Developed by IBM (systemR) in the 1970s
• Need for a standard since it is used by many vendors
• Standards:
  – SQL-86
  – SQL-89 (minor revision)
  – SQL-92 (major revision)
  – SQL-99 (major extensions, current standard)
Purposes of SQL

• Data Manipulation Language (DML)
  – Querying: SELECT-FROM-WHERE
  – Modifying: INSERT/DELETE/UPDATE

• Data Definition Language (DDL)
  – CREATE/ALTER/DROP
The SQL Query Language

• To find all 18 year old students, we can write:

```
SELECT * 
FROM Students S 
WHERE S.age=18
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>

• To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```
Querying Multiple Relations

• What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.sid AND E.grade = "A"
```

Given the following instances of Enrolled and Students:

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>
```

we get: ??
Querying Multiple Relations

• What does the following query compute?

```sql
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.sid AND E.grade = "A"
```

Given the following instances of Enrolled and Students:

Students

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Enrolled

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

we get:

<table>
<thead>
<tr>
<th>S.name</th>
<th>E.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Topology112</td>
</tr>
</tbody>
</table>
Creating Relations in SQL

• Creates the “Students” relation
  – the type (domain) of each field is specified
  – enforced by the DBMS whenever tuples are added or modified

• As another example, the “Enrolled” table holds information about courses that students take

```
CREATE TABLE Students
(sid CHAR(20),
 name CHAR(20),
 login CHAR(10),
 age INTEGER,
 gpa REAL)
```

```
CREATE TABLE Enrolled
(sid CHAR(20),
 cid CHAR(20),
 grade CHAR(2))
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Students

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

Enrolled
Destroying and Altering Relations

DROP TABLE Students

• Destroys the relation Students
  – The schema information and the tuples are deleted.

ALTER TABLE Students
  ADD COLUMN firstYear: integer

• The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a null value in the new field.
Adding and Deleting Tuples

• Can insert a single tuple using:

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

• Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```
Integrity Constraints (ICs)

- **IC**: condition that must be true for any instance of the database
  - e.g., domain constraints
  - ICs are specified when schema is defined
  - ICs are checked when relations are modified

- A **legal** instance of a relation is one that satisfies all specified ICs
  - DBMS will not allow illegal instances

- If the DBMS checks ICs, stored data is more faithful to real-world meaning
  - Avoids data entry errors, too!
Keys in a Database

- Key / Candidate Key
- Primary Key
- Super Key
- Foreign Key

- Primary key attributes are underlined in a schema
  - Person(pid, address, name)
  - Person2(address, name, age, job)
Primary Key Constraints

• A set of fields is a key for a relation if:
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key

• Part 2 false? A superkey
• If there are > 1 keys for a relation, one of the keys is chosen (by DBA = DB admin) to be the primary key
  – E.g., sid is a key for Students
  – The set {sid, gpa} is a superkey.

• Is there any possible benefit to refer to a tuple using primary key (than any key)?
Primary and Candidate Keys in SQL

• Possibly many candidate keys
  – specified using **UNIQUE**
  – one of which is chosen as the primary key.

• “For a given student and course, there is a single grade.”

```sql
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY ???)
```
Primary and Candidate Keys in SQL

- Possibly many candidate keys
  - specified using `UNIQUE`
  - one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid))
```
Primary and Candidate Keys in SQL

• Possibly many candidate keys
  – specified using UNIQUE
  – one of which is chosen as the primary key.

CREATE TABLE Enrolled
  (sid CHAR(20)
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))

CREATE TABLE Enrolled
  (sid CHAR(20)
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid),
   UNIQUE ???)

• “For a given student and course, there is a single grade.”

• VS.

• “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”
Primary and Candidate Keys in SQL

• Possibly many candidate keys
  – specified using `UNIQUE`
  – one of which is chosen as the primary key.

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid))

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY sid,
UNIQUE (cid, grade))

• “For a given student and course, there is a single grade.”

• `VS.`

• “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”
Primary and Candidate Keys in SQL

- Possibly many candidate keys
  - specified using `UNIQUE`
  - one of which is chosen as the primary key.

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid))
```

- “For a given student and course, there is a single grade.”

- vs.

- “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”

- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY sid, UNIQUE (cid, grade))
```

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid))
```
Foreign Keys, Referential Integrity

• **Foreign key**: Set of fields in one relation that is used to `refer` to a tuple in another relation
  – Must correspond to primary key of the second relation
  – Like a `logical pointer`

• **E.g. sid is a foreign key referring to Students:**
  – If all foreign key constraints are enforced, referential integrity is achieved
  – i.e., no dangling references
Foreign Keys in SQL

• Only students listed in the Students relation should be allowed to enroll for courses

CREATE TABLE Enrolled
    (sid CHAR(20), cid CHAR(20), grade CHAR(2),
     PRIMARY KEY (sid,cid),
     FOREIGN KEY (sid) REFERENCES Students )

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
Enforcing Referential Integrity

• Consider Students and Enrolled
  – sid in Enrolled is a foreign key that references Students.

• What should be done if an Enrolled tuple with a non-existent student id is inserted?
  – Reject it!

• What should be done if a Students tuple is deleted?
  – Three semantics allowed by SQL
    1. Also delete all Enrolled tuples that refer to it (cascade delete)
    2. Disallow deletion of a Students tuple that is referred to
    3. Set sid in Enrolled tuples that refer to it to a default sid
    4. (in addition in SQL): Set sid in Enrolled tuples that refer to it to a special value null, denoting `unknown’ or `inapplicable’

• Similar if primary key of Students tuple is updated
Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is **NO ACTION** (delete/update is rejected)
  - **CASCADE** (also delete all tuples that refer to deleted tuple)
  - **SET NULL** / **SET DEFAULT** (sets foreign key value of referencing tuple)

```sql
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid)
REFERENCES Students
ON DELETE CASCADE
ON UPDATE SET DEFAULT)
```
Where do ICs Come From?

• ICs are based upon the semantics of the real-world enterprise that is being described in the database relations

• Can we infer ICs from an instance?
  – We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.
  – An IC is a statement about all possible instances!
  – From example, we know name is not a key, but the assertion that sid is a key is given to us.

• Key and foreign key ICs are the most common; more general ICs supported too
Example Instances

• We will use these instances of the Sailors and Reserves relations in our examples

• If the key for the Reserves relation contained only the attributes sid and bid, how would the semantics differ?

<table>
<thead>
<tr>
<th>Sailor</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserves</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td></td>
</tr>
</tbody>
</table>
Basic SQL Query

SELECT [DISTINCT] <target-list>
FROM <relation-list>
WHERE <qualification>

- **relation-list**  A list of relation names
  - possibly with a “range variable” after each name
- **target-list**  A list of attributes of relations in relation-list
- **qualification**  Comparisons
  - (Attr op const) or (Attr1 op Attr2)
  - where op is one of =, <, >, <=, => combined using AND, OR and NOT
- **DISTINCT**  is an optional keyword indicating that the answer should not contain duplicates
  - Default is that duplicates are not eliminated!
Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of `<relation-list>`
  - Discard resulting tuples if they fail `<qualifications>`
  - Delete attributes that are not in `<target-list>`
  - If `DISTINCT` is specified, eliminate duplicate rows

- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers
**Example of Conceptual Evaluation**

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

### Step 1: Form **cross product** of Sailor and Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

### Sailor

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

### Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

---

Duke CS, Fall 2016
CompSci 516: Data Intensive Computing Systems
Example of Conceptual Evaluation

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid AND R.bid = 103
```

Step 2: Discard tuples that do not satisfy <qualification>
Example of Conceptual Evaluation

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Step 3: Select the specified attribute(s)**
A Note on “Range Variables”

• Really needed only if the same relation appears twice in the FROM clause
  – sometimes used as a short-name
• The previous query can also be written as:

```sql
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103
```

OR

```sql
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

It is good style, however, to use range variables always!
Find sailor ids who’ve reserved at least one boat

```
SELECT ???
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Find sailor ids who’ve reserved at least one boat

SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid

- Would adding DISTINCT to this query make a difference?
Find sailors who’ve reserved at least one boat

SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid

• Would adding `DISTINCT` to this query make a difference?
  – Note that if there are multiple bids for the same sid, you get multiple output tuples for the same sid
  – Without distinct, you get them multiple times

• What is the effect of replacing `S.sid` by `S.sname` in the `SELECT` clause?
  – Would adding `DISTINCT` to this variant of the query make a difference even if one sid reserves at most one bid?
Joins

• Condition/Theta-Join
• Equi-Join
• Natural-Join
• (Left/Right/Full) Outer-Join
### Condition/Theta Join

Select all from Sailors S, Reserves R where S.sid equals R.sid and age is greater than or equal to 40.

Form cross product, discard rows that do not satisfy the condition.
Equi Join

\[
\text{SELECT } * \\
\text{FROM } \text{Sailors S, Reserves R} \\
\text{WHERE } S.\text{sid} = R.\text{sid} \text{ and age } = 45
\]

A special case of theta join
Join condition only has equality predicate =

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
### Natural Join

A special case of equi join
Equality condition on ALL common predicates (sid)
Duplicate columns are eliminated

```sql
SELECT * 
FROM  Sailors S NATURAL JOIN Reserves R
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Outer Join

```sql
SELECT S.sid, R. bid
FROM Sailors S LEFT OUTER JOIN Reserves R
ON S.sid=R.sid
```

Preserves all tuples from the left table whether or not there is a match
if no match, fill attributes from right with null

Similarly RIGHT/FULL outer join

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
Expressions and Strings

```sql
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching
- **Find triples (of ages of sailors and two fields defined by expressions) for sailors**
  - whose names begin and end with B and contain at least three characters
- **LIKE** is used for string matching. `_` stands for any one character and `%` stands for 0 or more arbitrary characters
  - You will need these often
Find sid’s of sailors who’ve reserved a red or a green boat

- Assume a Boats relation (see book)
Find sid’s of sailors who’ve reserved a red or a green boat

- Assume a Boats relation
- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples
  - can themselves be the result of SQL queries
- If we replace **OR** by **AND** in the first version, what do we get?
- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND (B.color='red' OR B.color='green')
```

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='green'
```
Find sid’s of sailors who’ve reserved a red and a green boat.
Find sid’s of sailors who’ve reserved a red and a green boat

• **INTERSECT:** Can be used to compute the intersection of any two union-compatible sets of tuples.
  
  - Included in the SQL/92 standard, but some systems don’t support it

```sql
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
    AND S.sid=R2.sid AND R2.bid=B2.bid
    AND (B1.color='red' AND B2.color='green')
```

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='green'
```
Nested Queries

Find names of sailors who’ve reserved boat #103:

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN (SELECT  R.sid
                   FROM    Reserves R
                   WHERE   R.bid=103)
```

- A very powerful feature of SQL:
  - a WHERE/FROM/HAVING clause can itself contain an SQL query
- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation
  - For each Sailors tuple, check the qualification by computing the subquery

Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
    FROM Reserves R
    WHERE R.bid=103 AND S.sid=R.sid)
```

- **EXISTS** is another set comparison operator, like **IN**
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

```
SELECT S.sname 
FROM Sailors S  
WHERE UNIQUE (SELECT R.bid  
                FROM Reserves R  
                WHERE R.bid=103 AND S.sid=R.sid)
```

- If `UNIQUE` is used, and * is replaced by `R.bid`, finds sailors with at most one reservation for boat #103
  - `UNIQUE` checks for duplicate tuples
More on Set-Comparison Operators

- We’ve already seen \texttt{IN}, \texttt{EXISTS} and \texttt{UNIQUE}.
- Can also use \texttt{NOT IN}, \texttt{NOT EXISTS} and \texttt{NOT UNIQUE}.
- Also available: \texttt{op ANY}, \texttt{op ALL}, \texttt{op IN}
  - where \texttt{op} : \texttt{>, <, =, <=, >=}
- Find sailors whose rating is greater than that of some sailor called Horatio
  - similarly \texttt{ALL}

\begin{verbatim}
SELECT *  
FROM Sailors S  
WHERE S.rating > ANY (SELECT S2.rating  
FROM Sailors S2  
WHERE S2.sname='Horatio')
\end{verbatim}
Aggregated Operators

COUNT(*)
COUNT([DISTINCT] A)
SUM([DISTINCT] A)
AVG([DISTINCT] A)
MAX(A)
MIN(A)

SELECT COUNT(*)
FROM Sailors S

SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating=10

SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating) FROM Sailors S2)

SELECT COUNT(DISTINCT S.rating)
FROM Sailors S
WHERE S.sname=’Bob’

SELECT AVG(DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10

Check yourself:
What do these queries compute?
Motivation for Grouping

• So far, we’ve applied aggregate operators to all (qualifying) tuples
  – Sometimes, we want to apply them to each of several groups of tuples
• Consider: Find the age of the youngest sailor for each rating level
  – In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  – Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (need to replace $i$ by num):

\[
\text{SELECT MIN (S.age)} \\
\text{FROM Sailors S} \\
\text{WHERE S.rating = i}
\]

For $i = 1, 2, \ldots, 10$: 

Duke CS, Fall 2016

CompSci 516: Data Intensive Computing Systems
Queries With GROUP BY and HAVING

- The target-list contains
  - (i) attribute names
  - (ii) terms with aggregate operations (e.g., MIN (S.age))

- The attribute list (i) must be a subset of grouping-list
  - Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group
  - Here a group is a set of tuples that have the same value for all attributes in grouping-list
Conceptual Evaluation

- The cross-product of relation-list is computed
- Tuples that fail qualification are discarded
- "Unnecessary" fields are deleted
- The remaining tuples are partitioned into groups by the value of attributes in grouping-list
- The group-qualification is then applied to eliminate some groups
- Expressions in group-qualification must have a single value per group
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list
  - like "...GROUP BY bid, sid HAVING bid = 3"
- One answer tuple is generated per qualifying group
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

$$\text{SELECT} \ S.\text{rating}, \ \text{MIN} \ (S.\text{age}) \ AS \ minage$$
$$\text{FROM} \ \text{Sailors} \ S$$
$$\text{WHERE} \ S.\text{age} \geq 18$$
$$\text{GROUP BY} \ S.\text{rating}$$
$$\text{HAVING} \ \text{COUNT} \ (*) \ > 1$$

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 1: Form the cross product: FROM clause
(some attributes are omitted for simplicity)

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age \( \geq 18 \), for each rating with at least 2 such sailors.

Step 2: Apply WHERE clause

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 3: Apply GROUP BY according to the listed attributes

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 4: Apply HAVING clause

The *group-qualification* is applied to eliminate some groups.

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 5: Apply SELECT clause

Apply the aggregate operator
At the end, one tuple per group

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Null Values

• Field values in a tuple are sometimes
  – unknown, e.g., a rating has not been assigned, or
  – inapplicable, e.g., no spouse’s name
  – SQL provides a special value null for such situations.
Standard Boolean 2-valued logic

- True = 1, False = 0
- Suppose X = 5
  - (X < 100) AND (X >= 1) is T \(\land\) T = T
  - (X > 100) OR (X >= 1) is F \(\lor\) T = T
  - (X > 100) AND (X >= 1) is F \(\land\) T = F
  - NOT(X = 5) is \(\neg\)T = F

- Intuitively,
  - T = 1, F = 0
  - For V1, V2 \(\in\) \{1, 0\}
  - V1 \(\land\) V2 = MIN (V1, V2)
  - V1 \(\lor\) V2 = MAX(V1, V2)
  - \(\neg\)(V1) = 1 − V1
2-valued logic does not work for nulls

• Suppose rating = null, X = 5
• Is rating > 8 true or false?
• What about AND, OR and NOT connectives?
  – (rating > 8) AND (X = 5)?
• What if we have such a condition in the WHERE clause?
3-Valued Logic For Null

• TRUE (= 1), FALSE (= 0), UNKNOWN (= 0.5)
  – unknown is treated as 0.5

• Now you can apply rules from 2-valued logic!
  – For V1, V2 ∈ {1, 0, 0.5}
  – V1 ∧ V2 = MIN (V1, V2)
  – V1 ∨ V2 = MAX(V1, V2)
  – ~(V1) = 1 – V1

• Therefore,
  – NOT UNKNOWN = UNKNOWN
  – UNKNOWN OR TRUE = TRUE
  – UNKNOWN AND TRUE = UNKNOWN
  – UNKNOWN AND FALSE = FALSE
  – UNKNOWN OR FALSE = UNKNOWN
New issues for Null

• The presence of null complicates many issues. E.g.:
  – Special operators needed to check if value IS/IS NOT NULL
  – Be careful!
  – “WHERE X = NULL” does not work!
  – Need to write “WHERE X IS NULL”

• Meaning of constructs must be defined carefully
  – e.g., WHERE clause eliminates rows that don’t evaluate to true
  – So not only FALSE, but UNKNOWNs are eliminated too
  – very important to remember!

• But NULL allows new operators (e.g. outer joins)

• Arithmetic with NULL
  – all of +, -, *, / return null if any argument is null

• Can force ”no nulls” while creating a table
  – sname char(20) NOT NULL
  – primary key is always not null
## Aggregates with NULL

What do you get for

- `SELECT count(*)` from R1?
- `SELECT count(rating)` from R1?

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
Aggregates with NULL

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

• What do you get for
• SELECT count(*) from R1?
• SELECT count(rating) from R1?
• Ans: 3 for both
Aggregates with NULL

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R1

What do you get for
• SELECT count(*) from R1?
• SELECT count(rating) from R1?
• Ans: 3 for both

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R2

What do you get for
• SELECT count(*) from R2?
• SELECT count(rating) from R2?
Aggregates with NULL

R1

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

**What do you get for**
- SELECT count(*) from R1?
- SELECT count(rating) from R1?
- **Ans: 3 for both**

R2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

**What do you get for**
- SELECT count(*) from R2?
- SELECT count(rating) from R2?
- **Ans: First 3, then 2**
Aggregates with NULL

- **COUNT, SUM, AVG, MIN, MAX** (with or without DISTINCT)
  - Discards null values first
  - Then applies the aggregate
  - Except `count(*)`
- If only applied to null values, the result is null

### R2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

- SELECT `sum(rating)` from R2?
- Ans: 17

### R3

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>null</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>null</td>
<td>35</td>
</tr>
</tbody>
</table>

- SELECT `sum(rating)` from R3?
- Ans: null
Overview: General Constraints

- Useful when more general ICs than keys are involved
- There are also ASSERTIONS to specify constraints that span across multiple tables
- There are TRIGGERS too: procedure that starts automatically if specified changes occur to the DBMS
  - see additional slides at the end
Views

• A view is just a relation, but we store a definition, rather than a set of tuples

```
CREATE VIEW YoungActiveStudents (name, grade)
AS SELECT S.name, E.grade
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age<21
```

• Views can be dropped using the DROP VIEW command

• Views and Security: Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s)
  • the above view hides courses “cid” from E
Can create a new table from a query on other tables too

SELECT ... INTO ... FROM ... WHERE

SELECT S.name, E.grade
INTO YoungActiveStudents
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age<21
Summary

• SQL has a huge number of constructs and possibilities
  – You need to learn and practice it on your own
  – Given a problem, you should be able to write a SQL query and verify whether a given one is correct

• Pay attention to NULLs

• You will find “WITH” clause very useful!

  ```sql
  WITH Temp1 AS
      (SELECT ..... ..),
  Temp2 AS
      (SELECT ..... ..)
  SELECT X, Y
  FROM TEMP1, TEMP2
  WHERE....
  ```
Additional Examples for Practice

Check yourself
Rewriting INTERSECT Queries Using IN

Find sid’s of sailors who’ve reserved both a red and a green boat:

```
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
       AND S.sid IN (SELECT S2.sid
                       FROM   Sailors S2, Boats B2, Reserves R2
                       WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                              AND  B2.color='green')
```

• Similarly, `EXCEPT` queries re-written using `NOT IN`.
• To find names (not sid’s) of Sailors who’ve reserved both red and green boats, just replace `S.sid` by `S.sname` in `SELECT` clause
“Division” in SQL

Find sailors who’ve reserved all boats.

- Option 1:
- Option 2: Let’s do it the hard way, without EXCEPT:

```sql
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS ((SELECT B.bid
                   FROM Boats B)
                  EXCEPT
                   (SELECT R.bid
                    FROM Reserves R
                    WHERE R.sid = S.sid))
```

**Sailors S such that ...**

- **Option 1**: Sailors S such that there is no boat B... there is no boat B...
- **Option 2**: ...without ... ...without ...

**...a Reserves tuple showing S reserved B**

AND R.sid = S.sid)
Find name and age of the oldest sailor(s)

- The first query is illegal!
  - We’ll look into the reason a bit later, when we discuss GROUP BY

- The third query is equivalent to the second query
  - and is allowed in the SQL/92 standard, but is not supported in some systems

```sql
SELECT S.sname, MAX(S.age) FROM Sailors S
SELECT S.sname, S.age FROM Sailors S
WHERE S.age =
  (SELECT MAX(S2.age) FROM Sailors S2)
SELECT S.sname, S.age FROM Sailors S
WHERE (SELECT MAX(S2.age) FROM Sailors S2) = S.age
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors and with every sailor under 60.

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age $\geq 18$
GROUP BY S.rating
HAVING COUNT (*) $> 1$ AND EVERY (S.age $\leq 60$)

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>

What is the result of changing EVERY to ANY?
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 sailors between 18 and 60.

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18 AND S.age <= 60
GROUP BY S.rating
HAVING COUNT(*) > 1
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
For each red boat, find the number of reservations for this boat

```
SELECT  B.bid,  COUNT (*) AS scount
FROM    Sailors S, Boats B, Reserves R
WHERE   S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove B.color='red' from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?
Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

\[
\begin{align*}
\text{SELECT} & \quad \text{S.rating, MIN (S.age)} \\
\text{FROM} & \quad \text{Sailors S} \\
\text{WHERE} & \quad \text{S.age} > 18 \\
\text{GROUP BY} & \quad \text{S.rating} \\
\text{HAVING} & \quad 1 < (\text{SELECT COUNT (*)} \\
& \quad \text{FROM} \quad \text{Sailors S2} \\
& \quad \text{WHERE} \quad \text{S.rating}=\text{S2.rating})
\end{align*}
\]

- Shows `HAVING` clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if `HAVING` clause is replaced by:
  - `HAVING COUNT(*) > 1`
Find those ratings for which the average age is the minimum over all ratings

• Aggregate operations cannot be nested! **WRONG:**

\[
\begin{align*}
\text{SELECT } & S.\text{rating} \\
\text{FROM } & \text{Sailors } S \\
\text{WHERE } & S.\text{age} = (\text{SELECT MIN (AVG (S2.age)) FROM Sailors } S2)
\end{align*}
\]

• **Correct solution (in SQL/92):**

\[
\begin{align*}
\text{SELECT } & \text{Temp.rating, Temp.avgage} \\
\text{FROM } & (\text{SELECT } S.\text{rating, AVG (S.age) AS avgage} \\
\text{FROM } & \text{Sailors } S \\
\text{GROUP BY } & S.\text{rating}) \text{ AS Temp} \\
\text{WHERE } & \text{Temp.avgage} = (\text{SELECT MIN (Temp.avgage) FROM Temp})
\end{align*}
\]
Integrity Constraints (Review)

• An IC describes conditions that every legal instance of a relation must satisfy.
  – Inserts/deletes/updates that violate IC’s are disallowed.
  – Can be used to ensure application semantics (e.g., sid is a key), or prevent inconsistencies (e.g., sname has to be a string, age must be < 200)

• Types of IC’s: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  – Domain constraints: Field values must be of right type. Always enforced.
Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

```sql
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
    REFERENCING NEW TABLE NewSailors
    FOR EACH STATEMENT
    INSERT
    INTO YoungSailors(sid, name, age, rating)
    SELECT sid, name, age, rating
    FROM NewSailors N
    WHERE N.age <= 18
```